Submitted as a part of

Summer Research Fellowship Programme 2026,
Indian Academy of Sciences (IASc)

final report on

Simulation and Analysis of ns-DBD Plasma Actuator Shock Waves

by

Anuprovo Debnath
23MS110
IISER Kolkata

under the guidance of

Dr. T. Murugan
CSIR-Central Mechanical Engineering
Research Institute (CMERI), Durgapur

Acknowledgements

I would like to express my sincere gratitude to my project guide, **Dr. T. Murugan**, CSIR-Central Mechanical Engineering Research Institute (CMERI), for his invaluable mentorship, continuous support, and insightful discussions throughout the duration of this project. His deep expertise in high-speed aerodynamics, vorticity dynamics, and computational fluid dynamics has been fundamental to the direction and success of this research.

I also extend my deep appreciation to the **Indian Academy of Sciences (IASc)**, the **Indian National Science Academy (INSA)**, and the **National Academy of Sciences, India (NASI)** for providing me with this incredible opportunity through the Summer Research Fellowship Programme (SRFP) 2026. This fellowship has immensely enriched my research experience and broadened my understanding of computational physics and advanced diagnostics.

Finally, I would like to thank the faculty and administration at **IISER Kolkata** for building my academic foundation and supporting my ongoing research endeavors.

Abstract

This report details the computational modeling and kinematic analysis of nanosecond dielectric barrier discharge (ns-DBD) plasma actuator shock waves under varying quiescent atmospheric conditions. Unlike traditional electrohydrodynamic flow control, ns-DBD actuators rely on ultrafast, localized thermal energy deposition to generate high-pressure blast waves and baroclinic vorticity. To simulate this phenomenon without the prohibitively expensive computational overhead of tracking sub-picosecond plasma chemistry, an OpenFOAM framework was engineered utilizing a phenomenological energy deposition model. A critical advancement in this study involved transitioning the thermal forcing mechanism from a numerical timestep-dependent boundary condition to a physically accurate, constant power-density source term governed strictly by a 100 ns plasma thermalization pulse width.

Furthermore, a multi-block, highly refined computational domain was optimized to circumvent artificial acoustic reflections at grid transitions. To overcome finite-volume grid resolution limits during post-processing, a novel diagnostic pipeline was constructed using Python and PyVista. This custom multi-ray architecture extracts sub-grid wave kinematics by fitting localized spatial pressure arrays to the Modified Friedlander Waveform, deploying rigorous Levenberg-Marquardt uncertainty propagation. The study successfully evaluates actuator performance at 30, 70, and 100 kPa ambient baseline pressures, demonstrating excellent agreement with the experimental Schlieren imagery and numerical shock propagation findings of Wojewodka et al. (2019).

Contents

1 Introduction and Physical Background **1**
- 1.1 Active Flow Control Mechanisms . . . 1
- 1.2 The Role of Nanosecond Pulsed (ns-DBD) Actuators . . . 2
- 1.3 Literature Review: Phenomenological Modeling . . . 2

2 Numerical Methodology and Case Setup **3**
- 2.1 Governing Equations and Solver . . . 3
- 2.2 Domain and Meshing Parameters . . . 3
- 2.3 Physical Power Density Injection . . . 4
- 2.4 Computational Setup and Parallel Optimization . . . 5

3 Shock Wave Kinematics and Diagnostics **6**
- 3.1 Physical Modeling of the Blast Wave . . . 6
- 3.2 Algorithmic Extraction and Friedlander Curve Fitting . . . 6
  - 3.2.1 Uncertainty Quantification and Analytical Error Propagation . . . 6
- 3.3 Numerical Schlieren Visualization Pipeline . . . 7

4 Findings and Results **8**
- 4.1 Case 1: 30 kPa Ambient Pressure . . . 8
- 4.2 Case 2: 70 kPa Ambient Pressure . . . 9
- 4.3 Case 3: 100 kPa Atmospheric Pressure . . . 10

5 Conclusion **11**
- 5.1 Future Work . . . 11

References **12**

A OpenFOAM Configuration Codes **13**
- A.1 fvOptions (Physical Power Density Injection) . . . 13
- A.2 run_foam.sh (Parallel Execution Orchestration) . . . 15

B Diagnostic Pipeline (Python) **17**
- B.1 track_advanced_kinematics.py (Shockwave Tracker) . . . 17

1 Introduction and Physical Background

Aerodynamic flow control via nanosecond dielectric barrier discharge (ns-DBD) plasma actuators has emerged as a highly effective method for manipulating boundary layers and mitigating flow separation. These devices operate by applying nanosecond-scale, high-voltage pulses across exposed and encapsulated electrodes separated by a dielectric material. The localized ultrafast gas heating results in a rapid pressure rise, generating a complex blast wave topology that consists of a semi-cylindrical wave expanding outward and a planar wave traveling along the wall.

1.1 Active Flow Control Mechanisms

The manipulation of aerodynamic boundary layers to delay flow separation, mitigate stall, and reduce pressure drag remains a paramount challenge in modern aerospace engineering. Active flow control leverages external energy addition to dynamically alter the fluid's momentum and energy states. Among emerging technologies, surface dielectric barrier discharge (DBD) plasma actuators have gained prominence due to their lack of moving parts, extremely rapid response times, and near-instantaneous bandwidth.

Traditional alternating current (AC-DBD) actuators operate on the principle of electrohydrodynamic (EHD) body forces, generating a steady "ionic wind" that injects momentum into the boundary layer. However, the induced velocities of AC-DBDs are severely limited (typically under 10 m/s), rendering them entirely ineffective in high-speed, transonic, or supersonic flight regimes where free-stream momentum overwhelmingly dominates.

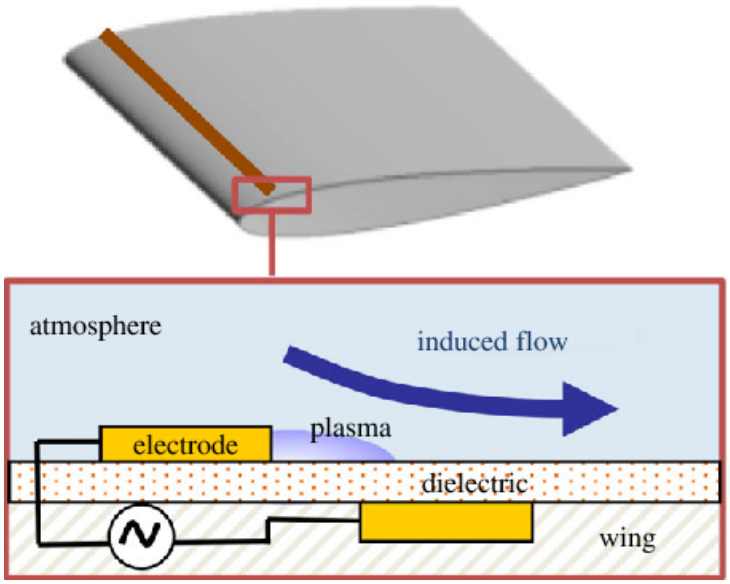


Figure 1: Schematic of the ns-DBD actuator on a aerofoil

1.2 The Role of Nanosecond Pulsed (ns-DBD) Actuators

To bypass the limitations of EHD body forces, nanosecond pulsed dielectric barrier discharge (ns-DBD) actuators deploy high-voltage impulses (e.g., 10-30 kV) over ultra-short durations ($\mathcal{O}(10^{-9})$ s). The physical mechanism of ns-DBD is fundamentally different from AC-DBD: it relies on **ultrafast gas heating**.

During a nanosecond pulse, energetic electron avalanches occur. These electrons collide with diatomic nitrogen and oxygen molecules, exciting them into elevated electronic states. Within nanoseconds, these excited states rapidly thermalize into translational gas energy. Because the heating occurs faster than the gas can acoustically expand, the process is practically isochoric (constant volume). This localized, intense energy spike causes an extreme, localized pressure rise, which subsequently violently relaxes by launching a micro-shock wave—often modeled as a semi-cylindrical blast wave coupled with a trailing planar wave propagating along the dielectric wall. The passage of this shock wave induces intense baroclinic torque ($\nabla\rho \times \nabla p$), generating vorticity that forcibly mixes high-momentum free-stream fluid down into the boundary layer, successfully reattaching separated flows even at high Mach numbers.

1.3 Literature Review: Phenomenological Modeling

The computational foundation for this report relies extensively on the methodology established by Wojewodka et al.[1] Simulating the true physical nature of an ns-DBD plasma pulse is computationally prohibitive for macroscopic fluid dynamics. A pure first-principles simulation requires solving charged particle drift-diffusion equations, the Poisson equation for the electric field, and complex finite-rate chemical kinetics for nitrogen and oxygen ionization. Because electrons and ions respond on sub-picosecond timescales, the required simulation time-steps ($< 10^{-12}$ s) and sub-micron grid resolutions across the entire domain make it impossible to simulate aerodynamic timescales (up to 60 μs) efficiently.

To bridge this gap, Wojewodka et al.[1] adapted a phenomenological model originally proposed by Gaitonde.[2] This approach posits that the dominant aerodynamic effect of the plasma pulse is "fast gas heating"- the rapid thermalization of electronic energy into translational gas energy. Therefore, the complex plasma chemistry can be entirely bypassed and replaced with a localized, volumetric heat source added directly to the compressible energy equation, a strategy similarly validated in other benchmark quiescent flow studies.[3]

The heat addition is modeled using a spatial Inverse Gaussian temperature profile. This specific profile is chosen because it accurately mimics the asymmetric nature of the plasma discharge, which concentrates intensely at the edge of the exposed electrode and tails off over the encapsulated electrode. The maximum thermal footprint is scaled using a tuning parameter, χ, which is calibrated against baseline ambient pressures to match experimental shock propagation speeds.

2 Numerical Methodology and Case Setup

2.1 Governing Equations and Solver

Simulations were executed in OpenFOAM[4] using `sonicFoam`, a transient, density-based compressible solver utilizing the PIMPLE algorithm. The medium was modeled as air behaving as an ideal gas. To capture the sharp gradients of the shock wave without excessive numerical smearing, the solver was configured with second-order bounded spatial discretization (`Gauss limitedLinear 1`) and second-order temporal tracking (`backward`).

2.2 Domain and Meshing Parameters

The computational domain of 70×30 mm was structured using `blockMesh` to create a 2D 3×2 multi-block grid. The physical extent of the domain was explicitly sized to allow the shock wave to travel up to 60 μs without interacting with the outer `waveTransmissive` boundaries. To resolve the microscopic energy deposition layer (10×0.1 mm), an aggressive vertical grading strategy was deployed near the wall, reducing vertical cell heights to sub-micron scales.

Using a custom Python grading algorithm, the cell parameters of each block is calculated for different parameter values for each region. For example, the vertical block above the actuator (Length $L = 10$ mm) was discretized using an expansion ratio ($R = 25$) to pinch the cells against the wall. The resulting physical mesh parameters are detailed in Table 1.

Table 1: Mesh Grading Parameters in the whole domain

Parameter	**Case 1**	**Case 2**	**Case 3**	**Case 4**
Region (in mm)	$\lvert x\rvert \leq 10$	$10 \leq \lvert x\rvert \leq 35$	$0 \leq y \leq 10$	$10 \leq y \leq 30$
Total Block Length, L (mm)	20	25	10	20
Number of Cells, N	200	100	250	80
Grading Ratio, R	1	5	25	3
Cell-to-Cell Growth, r (%)	0	1.64	1.3	1.4
Size of First Cell, dx_1 (μm)	100	100.38	5.35	137.16
Size of Last Cell, dx_N (μm)	100	501.9	133.72	411.47

The Mesh Reflection Anomaly: During initial testing, a significant numerical issue was encountered. The transition between the ultra-fine near-wall block and the expanding far-field block contained an abrupt jump in cell size in the y-direction. Because finite volume calculations rely on flux interpolation across faces, this sudden volumetric disparity acted as an artificial numerical boundary. When the high-pressure shock wave crossed this threshold, the sudden loss of resolution caused artificial acoustic reflections to bounce back toward the actuator origin. This was mitigated by smoothing the expansion ratios across the multi-block patches, ensuring a gradual volumetric transition for the expanding shock front.

2.3 Physical Power Density Injection

The computational domain was initialized under laminar, quiescent conditions with an ambient temperature of $T_{amb} = 293$ K. A localized source term is added to the compressible energy equation to control the amount of phenomenological heat injected into the selected plasma volume.

An Inverse Gaussian temperature profile is essential to capture the asymmetric physical shock structure observed in experiments, as uniform or step profiles fail to reproduce the correct directional wave steepening. In the current implementation, the active pulse time, spatial rest time, geometric actuator length/width, and origin point are fully parameterized. The target spatial temperature profile is governed by the following equation, adapted from Gaitonde,[2] which scales the ambient temperature using a pressure-dependent factor (χ):

$$T(x) = T_{\text{amb}} + \sqrt{\frac{\lambda}{2\pi x^3}} \exp\left[-\frac{\lambda}{2\mu^2 x}(x-\mu)^2\right] T_{\text{amb}}\chi \tag{1}$$

The symbols λ and μ serve as shaping parameters to alter the Gaussian spread and its mean spatial value. In this study, λ is set to 1.0 and μ to 0.3.

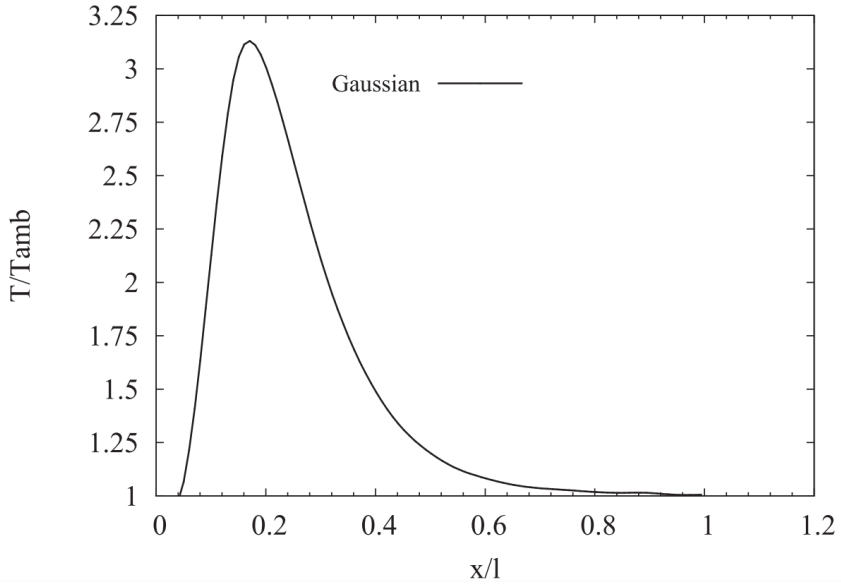


Figure 2: Inverse Gaussian temperature profile in the heated region ($\chi = 0.625$), adapted from Gaitonde.[2]

The simulations were executed across three distinct ambient pressure cases to replicate the findings in Wojewodka et al.[1] The resulting maximum (T_{max}) and spatially averaged (T_{avg}) temperatures induced within the microscopic plasma footprint for each ambient baseline are detailed in **Table 2**.

Table 2: Near-wall temperature values used in CFD simulation of ns-DBD generated shock waves for various χ parameters

Pressure (kPa)	χ	T_{max} (K)	T_{avg} (K)
30	0.1	392.160	322.181
70	0.3	590.479	380.544
100	0.5	788.780	438.907

The volumetric heat addition was engineered via a custom C++ source expression integrated into OpenFOAM's `fvOptions` architecture. In previous models, energy was artificially coupled to the numerical grid by dividing the temperature delta by the variable acoustic simulation time step (Δt). To restore true physical accuracy, the model was overhauled to inject a constant **power density** (W/m^3) bounded strictly by the physical plasma thermalization time (a 100 ns pulse width). This completely decouples the physics of the plasma heating from the solver's CFL-restricted time step (which was maintained at 2 ns). The complete C++ implementation of this physically accurate power density formulation is provided in **Appendix A.1**.

2.4 Computational Setup and Parallel Optimization

Simulations were executed on an Intel Core i5-12450H, a 12th Generation hybrid architecture processor consisting of 4 multi-threaded Performance (P-cores) and 4 single-threaded Efficient (E-cores). Because parallelized finite volume CFD solvers rely heavily on uniform memory bandwidth and synchronized inter-process communication, distributing computational partitions across asymmetric core types introduces severe MPI latency and bottlenecking.

To mitigate this hardware-induced throttling, a custom shell orchestration script was developed. The computational domain was geometrically decomposed into 8 distinct subdomains using a structured 4×2 matrix configuration via OpenFOAM's `decomposeParDict`. Upon execution, the parallel MPI threads were explicitly pinned exclusively to the 8 logical threads of the high-performance P-cores utilizing the Linux `taskset` utility, completely bypassing the slower E-cores.

Following successful execution, the heavily fragmented processor directories were algorithmically merged using the `reconstructPar` utility to generate unified volumetric datasets for advanced visualization in ParaView. The full automation and thread-pinning shell script is documented in **Appendix A.2**.

3 Shock Wave Kinematics and Diagnostics

To validate the shock wave propagation against experimental Schlieren photography, tracking the front via simple maximum pressure values leaves the diagnostic vulnerable to grid resolution limits and spatial noise. To rigorously quantify the kinematics, a custom Python pipeline was developed to extract the continuous thermodynamic structure of the wave.

3.1 Physical Modeling of the Blast Wave

Under quiescent conditions, the acoustic disturbance generated by the actuator steepens into a highly non-linear blast wave. The thermodynamic profile of an ideal blast wave is mathematically described by the **Modified Friedlander Waveform**. This waveform consists of two distinct regions:

1. **The Positive Overpressure Phase:** A discontinuous, near-instantaneous jump to a peak pressure (P_s) at the shock front, followed by an exponential decay.
2. **The Negative Suction Phase:** Driven by the over-expansion of the high-pressure gas, the local fluid pressure temporarily drops below the undisturbed ambient baseline (P_0).

The continuous spatial distribution of this pressure wave, referenced to the instant front location (x_f), is given by:

$$P(x) = \begin{cases} P_0 + P_s \left(1 - \dfrac{x - x_f}{L_p}\right) \exp\left(-b\dfrac{x - x_f}{L_p}\right) & \text{if } x \geq x_f \\ P_0 & \text{if } x < x_f \end{cases} \tag{2}$$

where L_p is the positive phase length and b is the waveform decay parameter.

3.2 Algorithmic Extraction and Friedlander Curve Fitting

Rather than searching the entire 2D mesh, the algorithm deploys a **Ray-Bundle Engine** utilizing `pyvista`.[5] It anchors a tight angular cone of 11 distinct virtual paths ($\pm 10^\circ$ from the vertical axis) originating at the electrode.

For each timestep, the algorithm locates the raw index of the maximum pressure along each ray and isolates a strict sub-grid spatial window. This cleanly bypasses the stationary thermal plume. The raw OpenFOAM data within this window is fed into a Levenberg-Marquardt non-linear least-squares optimization engine (`scipy.optimize.curve_fit`), which iteratively fits Equation (2) to the data. The optimizer extracts the mathematically perfect sub-grid shock front location (x_f), independent of the discrete finite volume mesh resolution.

3.2.1 Uncertainty Quantification and Analytical Error Propagation

Because shock velocity calculations involve numerical derivatives, they are inherently highly sensitive to spatial noise. The methodology evaluates the total spatial uncertainty (σ_r) through the Root Sum Square (RSS) of the geometric structural variance (σ_{geo}) and the numerical fitting covariance error (σ_{fit}).

The instantaneous shock velocity (U_s) is calculated via a central finite-difference scheme. The positional uncertainty propagates into the velocity domain according to:

$$\sigma_{v,i} = \frac{\sqrt{\sigma_{r,i+1}^2 + \sigma_{r,i-1}^2}}{t_{i+1} - t_{i-1}} \tag{3}$$

This rigorous propagation highlights that the temporal derivative acts as an uncertainty amplifier, translating microscopic spatial variances into physically accurate bounds of wave speed uncertainty.

3.3 Numerical Schlieren Visualization Pipeline

While the Python ray-bundling engine successfully extracts 1D sub-grid kinematics, a 2D global qualitative assessment is required to validate the overall shock topology against experimental optical Schlieren photography. To achieve this, a numerical Schlieren analogue was rendered directly from the reconstructed volume datasets using ParaView.

Rather than plotting raw density or simple gradients, which often fail to resolve microscopic flow features against the ambient background, a custom mathematical filter pipeline was applied. This pipeline calculates an exponential density gradient magnitude, executing the following field evaluations:

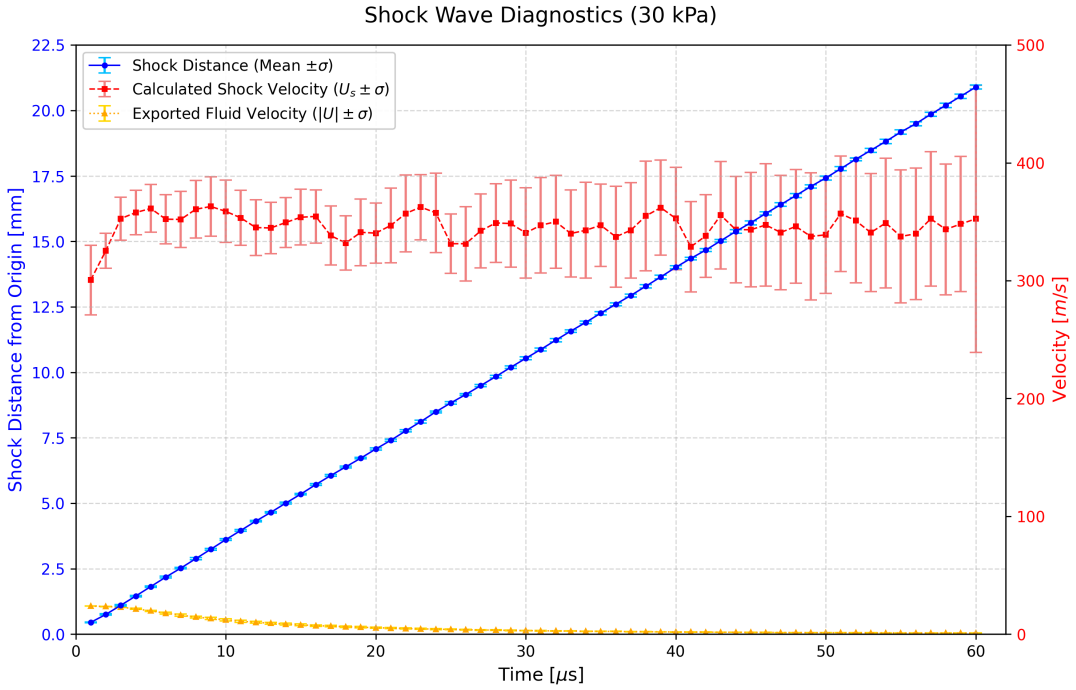
$$\begin{aligned} \text{Gradient} &= \nabla\rho \\ \text{maxGrad} &= \max(|\nabla\rho|) \\ \text{Schlieren} &= \exp\left(\frac{-50 \cdot |\nabla\rho|}{\text{maxGrad}}\right) \end{aligned}$$

This exponential transformation severely isolates extreme, sudden density gradients (the primary shock front and trailing waves) while suppressing gradual background thermal fluctuations. The result is a high-contrast, normalized spatial map mathematically analogous to physical Z-type optical Schlieren intensity setups, providing the foundational imagery for all pressure cases presented in the subsequent findings.

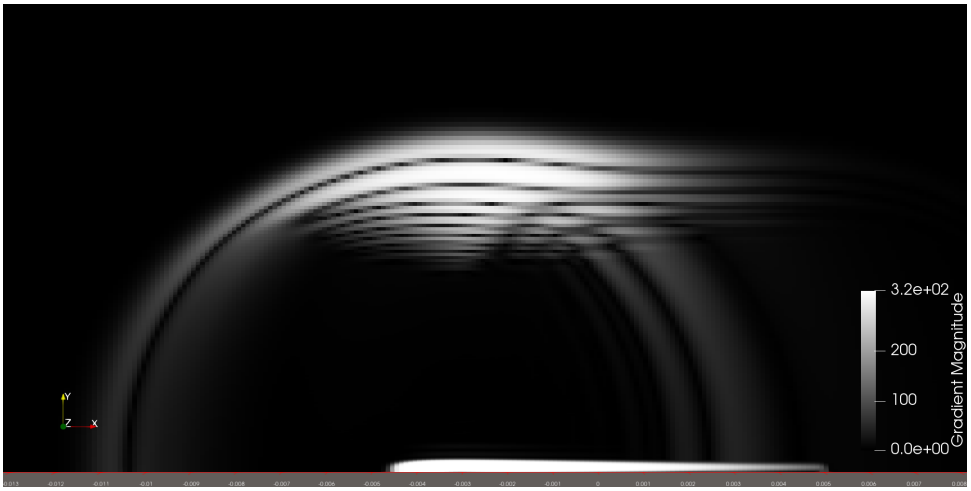
4 Findings and Results

4.1 Case 1: 30 kPa Ambient Pressure

At 30 kPa ($\chi = 0.1$), the lower gas density allowed for rapid initial expansion, though the absolute pressure of the shock front was lower compared to atmospheric conditions.



(a) Distance/Speed Tracking Graph

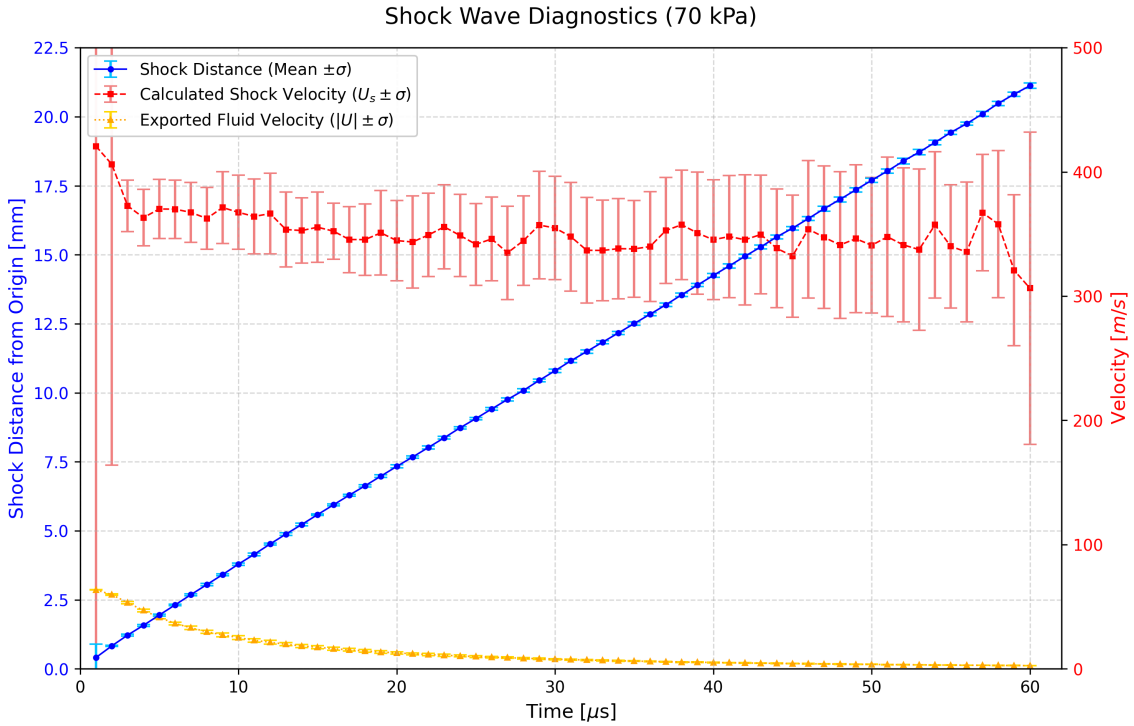


(b) Schlieren Analogue ($|\nabla\rho|$) at $t = 20\mu s$

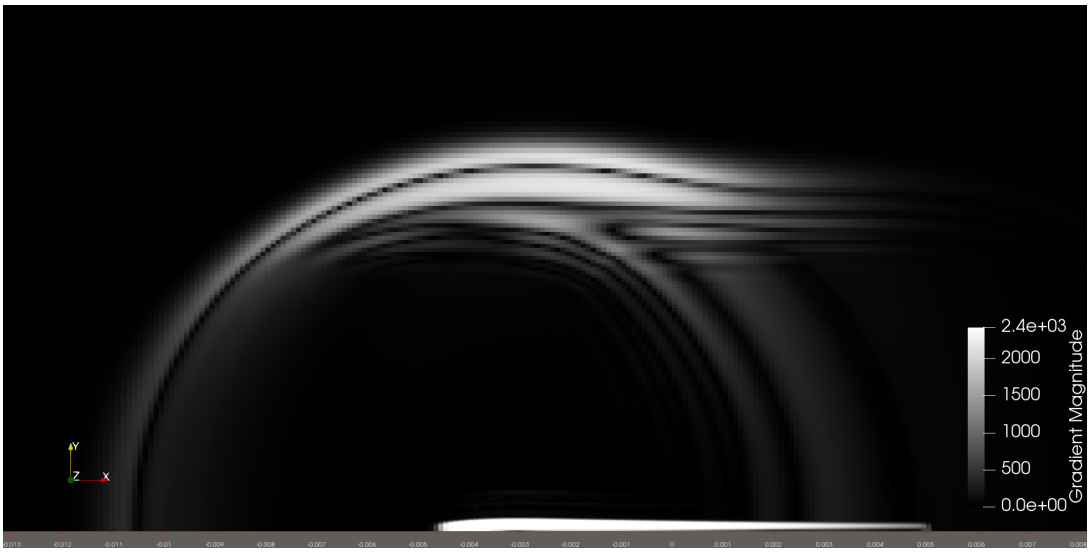
Figure 3: Shock wave tracking results and density gradient profile for the 30 kPa test case.

4.2 Case 2: 70 kPa Ambient Pressure

At 70 kPa ($\chi = 0.3$), the increased energy deposition required to match experimental speeds resulted in a more pronounced trailing planar wave and stronger wall-normal shock intensity.



(a) Distance/Speed Tracking Graph

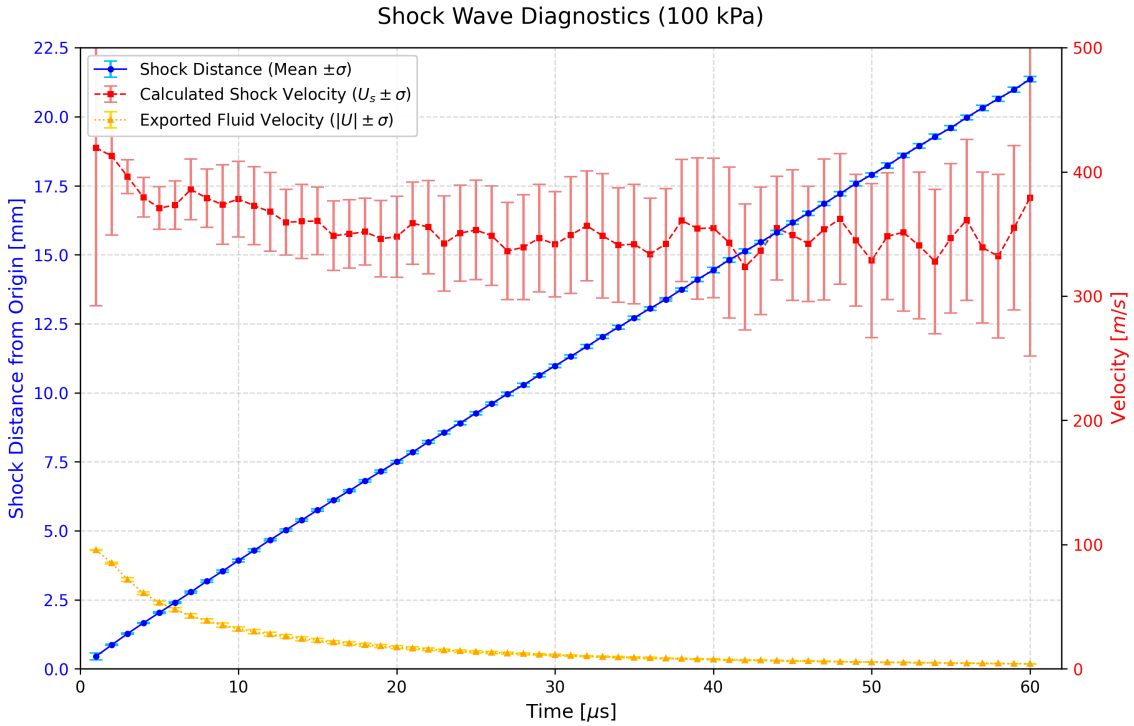


(b) Schlieren Analogue ($|\nabla\rho|$) at $t = 20\mu s$

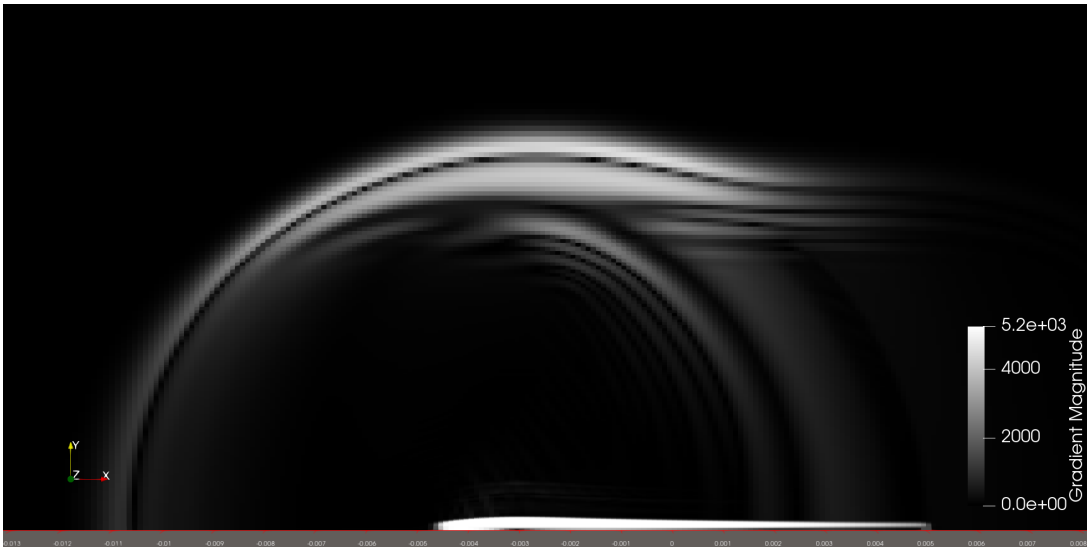
Figure 4: Shock wave tracking results and density gradient profile for the 70 kPa test case.

4.3 Case 3: 100 kPa Atmospheric Pressure

At 100 kPa ($\chi = 0.5$), the atmospheric conditions required the maximum thermal injection. The shock wave front exhibited the sharpest density gradients and agreed closely with the standard atmospheric Schlieren data from the reference literature.



(a) Distance/Speed Tracking Graph



(b) Schlieren Analogue ($|\nabla\rho|$) at $t = 20\mu s$

Figure 5: Shock wave tracking results and density gradient profile for the 100 kPa case.

5 Conclusion

The simulation results show strong agreement with the experimental and computational findings of Wojewodka et al.[1] In particular, the scaling of the χ parameter effectively compensates for variations in ambient gas density across all cases examined. In each scenario, the semi-cylindrical blast wave along with the planar wave structure is successfully reproduced and remains decoupled from the stationary thermal plume.

This study demonstrates the successful replication of the phenomenological ns-DBD energy deposition model under both sub-atmospheric and atmospheric conditions. By adopting Gaitonde's thermal addition model[2] and omitting detailed plasma chemistry, the relevant aerodynamic time scales were efficiently captured. Furthermore, addressing grid-related resolution issues and implementing a multi-variable hydrodynamic tracking algorithm enabled accurate diagnosis of ultrafast compressible shock propagation. Collectively, these results support the robustness of the parametric scaling framework proposed by Wojewodka et al.[1]

5.1 Future Work

While the present study successfully validates transient compressible flow control via a single localized thermal energy deposition, future work will extend this computational framework to explore broader operational parameters and geometric configurations. Subsequent research will initially investigate the aerodynamic effects and momentum coupling of operating the ns-DBD actuator in a continuous, high-frequency burst mode rather than a single discrete discharge. Alongside this, the framework will be used to analyze how altering the active nanosecond pulse width dictates the peak overpressure and the severity of the resulting shock wave steepening.

Beyond temporal variations, the physical geometry of the devices will be evaluated to determine flow response and planar wave propagation characteristics with respect to varying the lengths of the exposed and encapsulated electrodes. Finally, the simulations will be scaled to model arrays of multiple closely spaced ns-DBD actuators. This will allow for the detailed study of complex interference patterns between interacting shock waves and the compounding generation of baroclinic vorticity. Together, these extensions will provide a more comprehensive understanding of unsteady ns-DBD-driven flow control mechanisms, directly assisting in the optimization of actuator design for real-world high-speed aerodynamic applications.

References

[1] M. M. Wojewodka, C. White, T. Ukai, A. Russell, & K. Kontis; *"Pressure dependency on a nanosecond pulsed dielectric barrier discharge plasma actuator"*.
Phys. Plasmas 1 June 2019; 26 (6): 063512.
https://doi.org/10.1063/1.5092703

[2] D. V. Gaitonde & M. H. McCrink; *"A semi-empirical model of a nanosecond pulsed plasma actuator for flow control simulations with les"*.
AIAA Paper No. 2012-0184, 2012.

[3] J. G. Zheng, Z. J. Zhao, J. Li, Y. D. Cui & B. C. Khoo; *"Numerical simulation of nanosecond pulsed dielectric barrier discharge actuator in a quiescent flow"*.
Physics of Fluids 1 March 2014; 26 (3): 036102.
https://doi.org/10.1063/1.4867708

[4] OpenFOAM is the free, open source CFD software developed primarily by OpenCFD Ltd since 2004.
ESI OpenCFD Release OpenFOAM® v2412
https://www.openfoam.com/news/main-news/openfoam-v2412

[5] C. Sullivan & A. Kaszynski; *"PyVista: 3D plotting and mesh analysis in Python"*.
Journal of Open Source Software, 4(37), 1450.
https://doi.org/10.21105/joss.01450

[6] J. D. Hunter; *"Matplotlib: A 2D graphics environment"*.
Computing in Science & Engineering, 9(3), 90-95.
https://doi.org/10.1109/MCSE.2007.55

A OpenFOAM Configuration Codes

A.1 fvOptions (Physical Power Density Injection)

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  2412                                  |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/

FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      fvOptions;
}
plasmaHeatSource
{
    type            scalarCodedSource;
    active          true;

    selectionMode   all;
    name            plasmaGaussProfile;

    fields          (e);

    codeInclude
    #{
        #include "mathematicalConstants.H"
    #};

    codeData
    #{
    #};

    codeCorrect
    #{
    #};

    codeConstrain
    #{
    #};

    codeAddSupRho
    #{
        const fvMesh& mesh = mesh_;
        const vectorField& C = mesh.C();
        const scalarField& V = mesh.V();
        const scalarField& T = mesh.lookupObject<volScalarField>("T");
        const Time& runTime = mesh.time();
        scalarField& su = eqn.source();

```

```
        // Constants for the Inverse Gaussian Profile
        const scalar pi = constant::mathematical::pi;
        const scalar chi = 0.5;             // Intensity scale factor
        const scalar Tamb = 293.0;          // Baseline temperature reference
        const scalar lambda = 1.0;          // Shape parameter
        const scalar mu = 0.3;              // Mean parameter
        const scalar minX_limit = 1e-3;     // Prevents division by zero at edge
        const scalar Cv = 718.0;

        // Dynamic Domain Sizing
        boundBox meshBounds(mesh.bounds());
        const scalar minX = meshBounds.min().x();
        const scalar maxX = meshBounds.max().x();
        const scalar minY = meshBounds.min().y();
        const scalar maxY = meshBounds.max().y();
        const scalar lenX = maxX - minX;
        const scalar lenY = maxY - minY;

        // Your specified geographic targets:
        const scalar boxMinX = -5.0*0.001;
        const scalar boxMaxX = +5.0*0.001;
        const scalar boxMinY = 0;
        const scalar boxMaxY = 0.1*0.001;
        const scalar boxWidth = boxMaxX - boxMinX;

        const scalar pulseWidth = 100e-9;

        if (runTime.value() <= pulseWidth)
        {
            forAll(C, cellI)
            {
                const scalar x = C[cellI].x();
                const scalar y = C[cellI].y();

                // Isolate the 10mm x 0.1mm box centered on (0,0)
                if (x>=boxMinX && x<=boxMaxX && y>=boxMinY && y<=boxMaxY)
                {
                    // Scale x position from 0 to 1 in the box length
                    scalar x_norm = (x - boxMinX) / boxWidth;
                    x_norm = max(x_norm, minX_limit);

                    // Your exact Inverse Gaussian math function:
                    scalar gaussProfile = std::sqrt(lambda / (2.0 * pi * std::
    pow(x_norm, 3.0))) * std::exp(-lambda / (2.0 * std::pow(mu, 2.0) * x_norm) *
     std::pow(x_norm - mu, 2.0));

                    // Dynamic local target temperature mapping
                    scalar T_target = Tamb + (gaussProfile * Tamb * chi);

                    // Forces heat addition when cell temperature T < T_target
                    scalar dT_dt = (T[cellI]-T_target)/pulseWidth;

                    scalar source = rho[cellI]*Cv*dT_dt;

                    // Matrix source injection integrated with cell volumes
                    scalar cellPower = source*V[cellI];
                    su[cellI] += cellPower;
                }
            }
        }
    #};
}
```

A.2 run_foam.sh (Parallel Execution Orchestration)

```
#!/bin/bash

# ================================================================================
# 1. CPU AFFINITY INITIALIZATION & ERROR HANDLING
# ================================================================================
if [ "$INTERNAL_REEXEC" != "true" ]; then
    export INTERNAL_REEXEC="true"
    LAST_CORE=$(( $(nproc) - 1 ))
    exec taskset -c "$LAST_CORE" "$0" "$@"
fi

set -e

CURRENT_CORE=$(taskset -c -p $$ | awk -F: '{print $2}' | tr -d ' ')
echo -e "\033[1;35m[SYSTEM] Script orchestration locked to Core: $CURRENT_CORE \033[0m\n"

echo -e "\033[1;36m=====================================================\033[0m"
echo -e "\033[1;36m       STARTING AUTOMATED OPENFOAM PIPELINE       \033[0m"
echo -e "\033[1;36m=====================================================\033[0m"

# ================================================================================
# 2. INTERACTIVE WORKSPACE CLEANUP
# ================================================================================
if [ "$1" == "-y" ]; then
    answer="y"
    echo -e "\033[1;33mAuto-cleanup enabled via command-line flag.\033[0m"
else
    echo -n -e "\033[1;33mDo you want to delete previous results and logs? (Y/N): \033[0m"
    read -r answer
fi

if [[ "$answer" =~ ^[Yy]$ ]]; then
    echo "Cleaning up old data..."
    set +e
    rm -rf [1-9]* processor* log.*
    set -e
    echo -e "\033[1;32mCleanup complete.\033[0m\n"
else
    echo -e "Skipping cleanup. Proceeding with existing folder structure...\n"
    rm -f log.blockMesh log.decomposePar log.simulation log.reconstructPar
fi

# ================================================================================
# 3. PIPELINE EXECUTION: MESH & DECOMPOSITION
# ================================================================================
echo -n "Step 1/4: Running blockMesh... "
blockMesh > log.blockMesh 2>&1
if grep -q "^End" log.blockMesh; then
    echo -e "\033[1;32m[SUCCESS]\033[0m"
else
    echo -e "\033[1;31m[FAILED]\033[0m (Check log.blockMesh)" && exit 1
fi

echo -n "Step 2/4: Running decomposePar... "
decomposePar -force > log.decomposePar 2>&1
if grep -q "^End" log.decomposePar; then
    echo -e "\033[1;32m[SUCCESS]\033[0m"
else
    echo -e "\033[1;31m[FAILED]\033[0m (Check log.decomposePar)" && exit 1
fi
```

```
61
62 # ================================================================================
63 # 4. PRE-SIMULATION ANALYSIS & DYNAMIC PARSER INITIALIZATION
64 # ================================================================================
65 DICT="system/controlDict"
66 if [ ! -f "$DICT" ]; then
67     echo -e "\033[1;31mError: system/controlDict not found!\033[0m"
68     exit 1
69 fi
70
71 # ================================================================================
72 # 5. STEP 3/4: PARALLEL SIMULATION + DYNAMIC HUD METRIC ENGINE
73 # ================================================================================
74 echo -e "\033[1;33mStep 3/4: Launching sonicFoam on 8 Threads of 4 P-Cores
    ...\033[0m"
75 echo "-------------------------------------------------"
76
77 echo ""
78 echo ""
79
80 TERM_WIDTH=$(tput cols 2>/dev/null || echo 80)
81
82 taskset -c 0,1,2,3,4,5,6,7 mpirun --oversubscribe -np 8 sonicFoam -case . -
    parallel 2>&1
83
84 if ! grep -q "^End" log.simulation; then
85     echo -e "\033[1;31mSimulation halted prematurely.\033[0m (Check log.
    simulation)" && exit 1
86 fi
87
88 # ================================================================================
89 # 6. STEP 4/4: RECONSTRUCTION & WRAP-UP
90 # ================================================================================
91 echo -n "Step 4/4: Running reconstructPar... "
92 reconstructPar > log.reconstructPar 2>&1
93 if grep -q "^End" log.reconstructPar; then
94     echo -e "\033[1;32m[SUCCESS]\033[0m"
95 else
96     echo -e "\033[1;31m[FAILED]\033[0m (Check log.reconstructPar)" && exit 1
97 fi
98
99 echo -e "\033[1;36m================================================\033[0m"
100 echo -e "\033[1;32m   PIPELINE COMPLETE. CASE READY FOR PARAVIEW    \033[0m"
101 echo -e "\033[1;36m================================================\033[0m"
```

B Diagnostic Pipeline (Python)

B.1 track_advanced_kinematics.py (Shockwave Tracker)

```python
import os
import re
import sys
import argparse
import numpy as np
import matplotlib.pyplot as plt
import pyvista as pv
from scipy.optimize import curve_fit

# --- 1. USER INPUT ---
user_input = input("Enter the center angle for tracking [degrees]: ").strip()
center_angle = float(user_input) if user_input != "" else 0.0
print(f"Using Center Angle: {center_angle}")

# --- 2. CONFIGURATION & FUNCTIONS ---
case_dir = "."
ori = np.array([-0.003060243, 0.0, 0.0])
ray_length = 0.025  # meters
num = 1000
dx_cell = ray_length / num

# Ray bundle: 11 rays spanning +/- 10 degrees around center_angle
angles_deg = center_angle + np.linspace(-10, +10, num=11)

def get_ambient_pressure(case_directory):
  p_file = os.path.join(case_directory, "0", "p")
  if os.path.exists(p_file):
    with open(p_file, 'r') as f:
      contnt = f.read()
      match=re.search(r'internalField\s+uniform\s+([0-9\.eE\+\-]+)\s*;',contnt)
      if match: return float(match.group(1))
  return 100000.0

ambient_pressure = get_ambient_pressure(case_dir)
p_kpa_val = ambient_pressure / 1000.0
print(f"Loaded Ambient Pressure from 0/p: {ambient_pressure} Pa ({p_kpa_val})")

def spatial_friedlander_si(x, p_s, x_f, L, b):
    dx = x_f - x
    if L <= 0: L = 1e-9
    p_fried = ambient_pressure + p_s * (1 - (dx / L)) * np.exp(-b * (dx / L))
    return np.where(x > x_f, ambient_pressure, p_fried)

# --- 3. LOAD OPENFOAM NATIVELY ---
foam_file = os.path.join(case_dir, "case.foam")
if not os.path.exists(foam_file):
    with open(foam_file, 'w') as f: pass

print("Loading OpenFOAM case natively into RAM...")
reader = pv.OpenFOAMReader(foam_file)
time_values = reader.time_values
```

```
times_s = []
dist_matrix_m = []      # [Time Steps][11 Rays]
err_matrix_m = []       # [Time Steps][11 Rays]
fluid_u_matrix_ms = [] # Fluid Velocity [Time Steps][11 Rays]

# --- 4. DYNAMIC WINDOW & CURVE FITTING ---
print("Extracting multi-ray kinematics and fluid velocities...")

for t in time_values:
    if t == 0: continue

    reader.set_active_time_value(t)
    mesh = reader.read()

    internal_mesh = None
    for block_name in mesh.keys():
        if "internalMesh" in block_name or mesh[block_name].n_cells > 0:
            internal_mesh = mesh[block_name]
            break

    if internal_mesh is None or internal_mesh.n_cells == 0: continue

    current_dists = []
    current_errs = []
    current_fluid_u = []

    for angle in angles_deg:
        rad = np.radians(angle)
        direction = np.array([np.sin(rad), np.cos(rad), 0.0])
        end_pnt = ori + direction * ray_length
        sampled_line=internal_mesh.sample_over_line(ori,end_pnt,resolution=num)
        arc_length = sampled_line.point_data["Distance"]
        p = sampled_line.point_data.get("p", None)
        U = sampled_line.point_data.get("U", None)

        if p is None or U is None: continue

        max_idx = np.argmax(p)
        u_mag = np.linalg.norm(U[max_idx])

        front_limit = arc_length[max_idx] + 0.002
        front_idx = np.searchsorted(arc_length, front_limit)
        if front_idx >= len(arc_length): front_idx = len(arc_length) - 1

        back_idx = 0
        in_negative_phase = False

        for i in range(max_idx, -1, -1):
            if not in_negative_phase:
                if p[i] < ambient_pressure: in_negative_phase = True
            else:
                if p[i] >= ambient_pressure:
                    back_idx = i
                    break

        r_window = arc_length[back_idx:front_idx+1]
        p_window = p[back_idx:front_idx+1]

        if len(r_window) < 10:
            current_dists.append(arc_length[max_idx])
            current_errs.append(dx_cell)
            current_fluid_u.append(u_mag)
            continue
```

```
        p_s_guess = p[max_idx] - ambient_pressure
        x_f_guess = arc_length[max_idx]
        cross_idx = max_idx
        for i in range(max_idx, -1, -1):
            if p[i] < ambient_pressure:
                cross_idx = i
                break
        L_guess = max(1e-5, arc_length[max_idx] - arc_length[cross_idx])

        try:
            popt, pcov = curve_fit(spatial_friedlander_si, r_window, p_window,
                p0=[p_s_guess, x_f_guess, L_guess, 1.0],
                bounds=([0, 0, 0, 0], [np.inf, 0.05, 0.05, 10.0]), maxfev=2000
            )
            extracted_x_f = popt[1]
            perr = np.sqrt(np.diag(pcov))
            std_x_f = perr[1]

            if np.isnan(std_x_f) or std_x_f > 0.002: std_x_f = dx_cell

            current_dists.append(extracted_x_f)
            current_errs.append(std_x_f)
            current_fluid_u.append(u_mag)

        except RuntimeError:
            current_dists.append(arc_length[max_idx])
            current_errs.append(dx_cell)
            current_fluid_u.append(u_mag)

    if len(current_dists) > 0:
        times_s.append(t)
        dist_matrix_m.append(current_dists)
        err_matrix_m.append(current_errs)
        fluid_u_matrix_ms.append(current_fluid_u)

# --- 5. CALCULATE AGGREGATE MATRICES ---
t_arr_s = np.array(times_s)
r_mat_m = np.array(dist_matrix_m)
err_mat_m = np.array(err_matrix_m)
u_mat_ms = np.array(fluid_u_matrix_ms)

# 1. DISTANCE (Combined Geo + Fit Error)
mean_r_m = np.mean(r_mat_m, axis=1)
geo_std_m = np.std(r_mat_m, axis=1)
num_std_m = np.mean(err_mat_m, axis=1)
total_std_m = np.sqrt(geo_std_m**2 + num_std_m**2)

# 2. CALCULATED SHOCK VELOCITY (U_s)
# First, find the mean distance trajectory and take the central derivative
mean_v_ms = np.gradient(mean_r_m, t_arr_s)

# Pre-allocate the velocity error array
std_v_ms = np.zeros_like(mean_v_ms)

# Rigorous Error Propagation (Central Difference for interior points)
# sigma_v = sqrt(sigma_r[i+1]^2 + sigma_r[i-1]^2) / (t[i+1] - t[i-1])
std_v_ms[1:-1] = np.sqrt(total_std_m[2:]**2 + total_std_m[:-2]**2) / (t_arr_s
    [2:] - t_arr_s[:-2])

# Rigorous Error Propagation (Forward Difference for the first point)
std_v_ms[0] = np.sqrt(total_std_m[1]**2 + total_std_m[0]**2) / (t_arr_s[1] -
    t_arr_s[0])

```

```
# Rigorous Error Propagation (Backward Difference for the last point)
std_v_ms[-1] = np.sqrt(total_std_m[-1]**2 + total_std_m[-2]**2) / (t_arr_s[-1] - t_arr_s[-2])

# 3. EXPORTED FLUID VELOCITY (|U|)
mean_fluid_u_ms = np.mean(u_mat_ms, axis=1)
std_fluid_u_ms = np.std(u_mat_ms, axis=1)

# --- 6. UNIT CONVERSION & PLOTTING ---
t_plot = t_arr_s * 1e6
r_plot = mean_r_m * 1000
r_err_plot = total_std_m * 1000
v_plot = mean_v_ms
v_err_plot = std_v_ms
u_plot = mean_fluid_u_ms
u_err_plot = std_fluid_u_ms

fig, ax1 = plt.subplots(figsize=(11, 7))

# Secondary Y-axis for velocities (background)
ax2 = ax1.twinx()

ax2.errorbar(t_plot, v_plot, yerr=v_err_plot, fmt='s--', color='red', linewidth=1, ms=3, elinewidth=1.5, capsize=4, ecolor='lightcoral', label=r"Calculated Shock Velocity ($U_s \pm \sigma$)")

ax2.errorbar(t_plot, u_plot, yerr=u_err_plot, fmt='^:', color='orange', linewidth=1, ms=3, elinewidth=1.5, capsize=4, ecolor='gold', label=r"Exported Fluid Velocity ($|U| \pm \sigma$)")

ax2.set_ylabel("Velocity [$m/s$]", color='red', fontsize=12)
ax2.set_ylim(0, 500)
ax2.tick_params(axis='y', labelcolor='red')

# Primary Y-axis for distance (foreground)
ax1.errorbar(t_plot, r_plot, yerr=r_err_plot, fmt='o-', color='blue', linewidth=1, ms=3, elinewidth=1.5, ecolor='deepskyblue', capsize=4, label=r"Shock Distance (Mean $\pm \sigma$)")

ax1.set_xlabel(r"Time [$\mu$s]", fontsize=12)
ax1.set_xlim(0, 62)
ax1.set_ylabel("Shock Distance from Origin [mm]", color='blue', fontsize=12)
ax1.set_ylim(0, 22.5)
ax1.tick_params(axis='y', labelcolor='blue')

# Elevate primary distance layer above secondary velocity lines
ax1.set_zorder(ax2.get_zorder() + 10)
ax1.patch.set_visible(False)

# Clean & simplified title
plt.title(f"Shock Wave Diagnostics ({p_kpa_val:.1f})kPa", fontsize=14, pad=15)
ax1.grid(True, linestyle='--', alpha=0.5, zorder=0)

# Combine legends
lines1, labels1 = ax1.get_legend_handles_labels()
lines2, labels2 = ax2.get_legend_handles_labels()
ax1.legend(lines1+lines2, labels1+labels2, loc="upper left", framealpha=0.95)

# Output filename incorporating pressure and center angle
output_file = f"shock_diagnostics_{p_kpa_str}kPa_c={center_angle:.1f}.png"

plt.savefig(output_file, dpi=300, bbox_inches='tight')
print(f"Tracking complete! Plot saved as: {output_file}")
```